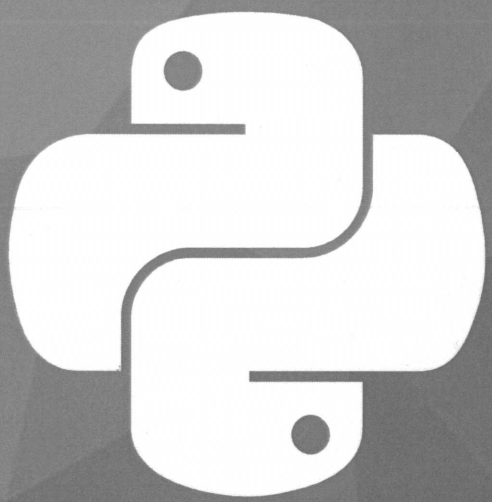
PYTHON

for complete beginners



Dr. Martin Jones

Table of Contents

2: Working with strings and numbers

Why are we so interested in working with text?

Open the first page of a book about learning Python[1], and the chances are that the first examples of code you'll see involve **numbers**. There's a good reason for that: numbers are generally simpler to work with than text – there are not too many things you can do with them (once you've got basic arithmetic out of the way) and so they lend themselves well to examples that are easy to understand. It's also a pretty safe bet that the average person reading a programming book is doing so because they need to do some number crunching.

The numbers-first approach makes a lot of sense for readers coming from a computer science or maths background, but it's not so good for the rest of us. Concentrating just on manipulating numbers leads to a lot of fairly boring examples and exercises that feel like we're just using Python as a calculator. So for this book, we're going to do things a bit differently: we'll start by playing around with a mixture of both text and numbers. This will allow us to look at more interesting examples and exercises, and to get more of a feel for how the language works.

I've hinted above that Python treats numbers and text differently. That's an important idea, and one that we'll return to in more detail later. For now, I want to introduce an important piece of jargon – the word *string*. String is the word we use to refer to a bit of text in a computer program (it just means a string of characters). From this point on we'll use the word *string* when we're talking about computer code.

1 Or indeed, any other programming language

Printing a message to the screen

The first thing we're going to learn is how to print a message to the screen. When we talk about *printing* text inside a computer program, we are not talking about producing a document on a printer. The word "print" is used for any occasion when our program outputs some text – in this case, the output is displayed in your terminal window or the output window of your IDE. Here's a line of Python code that will cause a friendly message to be printed. Quick reminder: solid lines indicate Python code, and dotted lines indicate the output.

```
print("Hello world")
```

hello_world.py

Let's take a look at the various bits of this line of code, and give some of them names:

The whole line is called a *statement*.

`print()` is the name of a *function*. The function tells Python, in vague terms, what we want to do – in this case, we want to print some text. The function name is always[1] followed by parentheses.

The bits of text inside the parentheses are called the *arguments* to the function. In this case, we just have one argument (later on we'll see examples of functions that take more than one argument, in which case the arguments are separated by commas).

The arguments tell Python what we want to do more specifically – in this case, the argument tells Python exactly what it is we want to print: a friendly greeting.

1 This is not strictly true, but it's easier to just follow this rule than worry about the exceptions.

Assuming you've followed the instructions in chapter 1 and set up your Python environment, type the line of code above into your favourite text editor or IDE, save it, and run it. You should see a single line of output like this:

```
Hello world
```

Quotes are important

In normal writing, we only surround a bit of text in quotes when we want to show that they are being spoken by somebody. In Python, however, strings are **always** surrounded by quotes. That is how Python is able to tell the difference between the instructions (like the function name) and the data (the thing we want to print). We can use either single or double quotes for strings – Python will happily accept either. The following two statements behave exactly the same:

```
print("Hello world")
print('Hello world')
```

different_quotes.py

Let's take a look at the output to prove it[1]:

```
Hello world
Hello world
```

You'll notice that the output above doesn't contain quotes – they are part of the code, not part of the string itself. If we **do** want to include quotes in

1 From this point on, I won't tell you to create a new file, enter the text, and run the program for each example – I will simply show you the output – but I encourage you to try the examples yourself.

the output, the easiest thing to do[1] is use the other type of quotes for surrounding the string:

```
print("She said, 'Hello world'")
print('He said, "Hello world"')
```

printing_quotes.py

The above code will give the following output:

```
She said, 'Hello world'
He said, "Hello world"
```

Be careful when writing and reading code that involves quotes – you have to make sure that the quotes at the beginning and end of the string match up.

Use comments to annotate your code

Occasionally, we want to write some text in a program that is for humans to read, rather than for the computer to execute. We call this type of line a *comment*. To include a comment in your source code, start the line with a hash symbol[2]:

```
# this is a comment, it will be ignored by the computer
print("Comments are very useful!")
```

comment.py

1 The alternative is to place a backslash character (\) before the quote – this is called *escaping* the quote and will prevent Python from trying to interpret it.

2 This symbol has many names – you might know it as number sign, pound sign, octothorpe, sharp (from musical notation), cross, or pig-pen.

You're going to see a lot of comments in the source code examples in this book, and also in the solutions to the exercises. Comments are a very useful way to document your code, for a number of reasons:

- You can put the explanation of what a particular bit of code does right next to the code itself. This makes it much easier to find the documentation for a line of code that is in the middle of a large program, without having to search through a separate document.
- Because the comments are part of the source code, they can never get mixed up or separated. In other words, if you are looking at the source code for a particular program, then you automatically have the documentation as well. In contrast, if you keep the documentation in a separate file, it can easily become separated from the code.
- Having the comments right next to the code acts as a reminder to update the documentation whenever you change the code. The only thing worse than undocumented code is code with old documentation that is no longer accurate!

Don't make the mistake, by the way, of thinking that comments are only useful if you are planning on showing your code to somebody else. When you start writing your own code, you will be amazed at how quickly you forget the purpose of a particular section or statement. If you are working on a solution to one of the exercises in this book on Friday afternoon, then come back to it on Monday morning, it will probably take you quite a while to pick up where you left off.

Comments can help with this problem by giving you hints about the purpose of code, meaning that you spend less time trying to understand your old code, thus speeding up your progress. A side benefit is that writing a comment for a bit of code reinforces your understanding at the

time you are doing it. A good habit to get into is writing a quick one line comment above any line of code that does something interesting:

```
# print a friendly greeting
print("Hello world")
```

You'll see this technique used a lot in the code examples in this book, and I encourage you to use it for your own code as well.

Error messages and debugging

It may seem depressing early in the book to be talking about errors! However, it's worth pointing out at this early stage that **computer programs almost never work correctly the first time**. Programming languages are not like natural languages – they have a very strict set of rules, and if you break any of them, the computer will not attempt to guess what you intended, but instead will stop running and present you with an error message. You're going to be seeing a lot of these error messages in your programming career, so let's get used to them as soon as possible.

Forgetting quotes

Here's one possible error we can make when printing a line of output – we can forget to include the quotes:

```
print(Hello world)
```

missing_quotes.py

This is easily done, so let's take a look at the output we'll get if we try to run the above code[1]:

1 The output that you see might be very slightly different from this, depending on a bunch of

```
$ python error.py  File "error.py"❶, line 1❷
    print(Hello world)❸
                     ^
SyntaxError: invalid syntax
```

We can see that the name of the Python file is `error.py`❶ and that the error occurs on the first line of the file❷. Python's best guess at the location of the error is just before the close parentheses❸. Depending on the type of error, this can be wrong by quite a bit, so don't rely on it too much!

The type of error is a `SyntaxError`, which mean that Python can't understand the code – it breaks the rules in some way (in this case, the rule that strings must be surrounded by quotation marks). We'll see different types of errors later in this book.

Spelling mistakes

What happens if we misspell the name of the function?:

```
prin("Hello world")
```

spelling.py

We get a different type of error – a `NameError` – and the error message is a bit more helpful:

```
    prin("Hello world")❶
NameError: name 'prin' is not defined❷
```

factors like your operating system and the exact version of Python you are using.

This time, Python doesn't try to show us where on the line the error occurred, it just shows us the whole line❶. The error message tells us which word Python doesn't understand❷, so in this case, it's quite easy to fix.

Splitting a statement over two lines

What if we want to print some output that spans multiple lines? For example, we want to print the word "Hello" on one line and then the word "World" on the next line – like this:

```
Hello
World
```

We might try putting a new line in the middle of our string like this:

```
print("Hello
World")
```

but that won't work and we'll get the following error message:

```
  File "error.py", line 1❶
    print("Hello
                ^
SyntaxError: EOL while scanning string literal❷
```

Python finds the error when it gets to the end of the first line of code❶. The error message❷ is a bit more cryptic than the others. *EOL* stands for End Of Line, and *string literal* means a string in quotes. So to put this error message in plain English: *"I started reading a string in quotes, and I got to the end of the line before I came to the closing quotation mark"*

If splitting the line up doesn't work, then how do we get the output we want.....?

Printing special characters

The reason that the code above didn't work is that Python got confused about whether the new line was part of the *string* (which is what we wanted) or part of the *source code* (which is how it was actually interpreted). What we need is a way to include a new line as part of a string, and luckily for us, Python has just such a tool built in. To include a new line, we write a backslash followed by the letter n – Python knows that this is a special character and will interpret it accordingly. This special character is called a *newline*. Here's the code which prints "Hello world" across two lines:

```
# how to include a new line in the middle of a string
print("Hello\nworld")
```

print_newline.py

Notice that there's no need for a space before or after the newline. This newline character will become very important in the next chapter when we start reading data from files.

There are a few other useful special characters as well, all of which consist of a backslash followed by a letter. The only ones which you are likely to need for the exercises in this book are the *tab* character (`\t`) and the *carriage return* character (`\r`). The tab character can sometimes be useful when writing a program that will produce a lot of output. The carriage return character works a bit like a newline in that it puts the cursor back to the start of the line, but doesn't actually start a new line, so you can use it to overwrite output – this is sometimes useful for long running programs.

Storing strings in variables

OK, we've been playing around with the `print()` function for a while; let's introduce something new. We can take a string and assign a name to it using an equals sign – we call this a *variable*:

```
# store a greeting in the variable called my_greeting
my_greeting = "Hello!"
```

The variable `my_greeting` now points to the string `"Hello!"`. We call this *assigning* a variable, and once we've done it, we can use the variable name instead of the string itself – for example, we can use it in a `print()` statement[1]:

```
# store a greeting in the variable called my_greeting
my_greeting = "Hello!"

# now print the greeting
print(my_greeting)
```

print_variable.py

Notice that when we use the variable in a `print()` statement, we don't need any quotation marks – the quotes are part of the string, so they are already "built in" to the variable `my_greeting`. Also notice that this example includes a blank line to separate the different bits and make it easier to read. We are allowed to put as many blank lines as we like in our programs when writing Python – the computer will ignore them.

We can change the value of a variable as many times as we like once we've created it:

1 If it's not clear why this is useful, don't worry – it will become much more apparent when we look at some longer examples.

```
my_greeting = "Hello!"
print(my_greeting)

# change the value of my_greeting
my_greeting = "Hi, friend!"
```

Here's a very important point that trips many beginners up: variable names are **arbitrary** – that means that we can pick **whatever we like** to be the name of a variable. So our code above would work in exactly the same way if we picked a different variable name:

```
# store a greeting in the variable banana
banana = "Hello!"

# now print the greeting
print(banana)
```

What makes a good variable name? Generally, it's a good idea to use a variable name that gives us a clue as to what the variable refers to. In this example, `my_greeting` is a good variable name, because it tells us what kind of information is stored in the variable. Conversely, `banana` is a bad variable name, because it doesn't really tell us anything about the value that's stored. As you read through the code examples in this book, you'll get a better idea of what constitutes good and bad variable names.

This idea – that names for things are arbitrary, and can be anything we like – is a theme that will occur many times in this book, so it's important to keep it in mind. Occasionally you will see a variable name that **looks like** it has some sort of relationship with the value it points to:

```
my_file = "my_file.txt"
```

but don't be fooled! Variable names and strings are separate things.

I said above that variable names can be anything we want, but it's actually not quite that simple – there are some rules we have to follow. We are only allowed to use letters, numbers, and underscores, so we can't have variable names that contain odd characters like £, ^ or %. We are not allowed to start a name with a number (though we can use numbers in the middle or at the end of a name). Finally, we can't use a word that's already built in to the Python language like "print".

It's also important to remember that variable names are case sensitive, so `my_greeting`, `MY_GREETING` and `My_Greeting` are all separate variables. Technically this means that you could use all three names in a Python program to store different values, but please don't do this – it is very easy to become confused when you use very similar variable names.

Tools for manipulating strings

Now we know how to store and print strings, we can take a look at a few of the facilities that Python has for manipulating them. In the exercises at the end of this chapter, we'll look at how we can use multiple different tools together in order to carry out more complex operations.

Concatenation

We can concatenate (stick together) two strings using the + symbol[1]. This symbol will join together the string on the left with the string on the right:

```
my_greeting = "Hel" + "lo"
print(my_greeting)
```

print_concatenated.py

1 We call this the *concatenation operator*.

Let's take a look at the output:

```
Hello
```

In the above example, the things being concatenated were strings, but we can also use variables that point to strings:

```
first_part = "Hel"
my_greeting = first_part + "lo"
# my_greeting is now "Hello"
```

We can even join multiple strings together in one go:

```
first_word = "Hello"
second_word = "World"
my_greeting = first_word + " " + second_word
# my_greeting is now "Hello World"
```

Notice in this last example how the middle string is a single space character. Although in everyday writing we think of a space as just a gap between two words, in programming a space is a character just like any other.

It's important to realize that the result of concatenating two strings together is itself a string. So it's perfectly OK to use a concatenation inside a `print()` statement, for example:

```
print("Hello" + " " + "world")
```

As we'll see in the rest of the book, using one tool inside another is quite a common thing to do in Python.

Finding the length of a string

Another useful built in tool in Python is the `len()` function (`len` is short for length). Just like `print()`, `len()` takes a single argument (take a quick look back at when we were discussing the `print()` function for a reminder about what arguments are) which is a string. However, the behaviour of the `len()` function is quite different. Instead of outputting text to the screen, `len()` outputs a value that can be stored – we call this the *return value*. In other words, if we write a program that uses `len()` to calculate the length of a string, the program will run but we won't see any output:

```
# this line doesn't produce any output
len("Hello World")
```

If we want to actually use the return value, we need to store it in a variable, and then do something useful with it (like printing it):

```
greeting_length = len("Hello world!")
print(greeting_length)
```

print_length.py

There's another interesting thing about the `len()` function: the result (or *return value*) is not a string, it's a number. This is a very important idea so I'm going to write it out in bold: **Python treats strings and numbers differently.**

We can see that this is the case if we try to concatenate together a number and a string. Consider this short program which calculates the length of my name and then prints out a message telling us the length:

```
# store my name in a variable
my_name = "Martin"

# calculate the length of my name and store it in a variable
name_length = len(my_name)

# print a message telling us the length of the name
("The length of the name is " + name_length)
```

When we try to run this program, we get the following error:

```
    print("The length of the name is " + name_length)
TypeError: cannot concatenate 'str' and 'int' objects❶
```

The error message❶ is short but informative: "`cannot concatenate 'str' and 'int' objects`". Python is complaining that it doesn't know how to concatenate a **string** (which it calls `str` for short) and a **number** (which it calls `int` – short for integer). Strings and numbers are examples of *data types* – different kinds of information that can exist inside a program.

Happily, Python has a built in solution to this problem – a function called `str()` which turns a number[1] into a string so that we can print it. Here's how we can modify our program to use it – I've removed the comments from this version to make it a bit more compact:

```
my_name = "Martin"
name_length = len(my_name)
print("The length of the name is " + str(name_length))
```

print_name_length.py

1 Or a value of any non-string type, but we'll come to that later.

The only thing we have changed is that we've replaced `name_length` with `str(name_length)` inside the `print()` statement[2]. Notice that because we're using one function (`str()`) inside another function (`print()`), our statement now ends with two closing parentheses.

Let's take a moment to refresh our memory of all the new terms we've learned by writing out what we need to know about the `str()` function:

> `str()` is a *function* which takes one *argument* (whose type is *number*), and *returns* a value (whose type is *string*) representing that number.

If you're unsure about the meanings of any of the words in italics, skip back to the earlier parts of this chapter where we discussed them. Understanding how types work is key to avoiding many of the frustrations which new programmers typically encounter, so make sure the idea is clear in your mind before moving on with the rest of this book.

Sometimes we need to go the other way – we have a string that we need to turn into a number. The function for doing this is called `int()`, which is short for integer. It takes a string as its argument and returns a number:

```
number = 3 + int('4')
# number is now 7
```

We won't need to use `int()` for a while, but once we start reading information from files later on in the book it will become very useful.

Changing case

We can convert a string to lower case by using a new type of tool – a *method* that belongs to strings. A *method* is like a *function*, but instead of being built in to the Python language, it belongs to a particular *type*. The

2 If you experiment with some of the code here, you might discover that you can also print a number directly without using `str` – but only if you don't try to concatenate it.

method we are talking about here is called `lower()`, and we say that it belongs to the *string* type. Here's how we use it:

```
my_name = "MARTIN"
# print my_name in lower case
print(my_name.lower())
```

print_lower.py

Notice how using a method looks different to using a function. When we use a function like `print()` or `len()`, we write the function name first and the arguments go in parentheses:

```
print("Hello")
len(my_name)
```

When we use a method, we write the name of the variable first, followed by a period, then the name of the method, then the method arguments in parentheses. For the example we're looking at here, `lower()`, there is no argument, so the opening and closing parentheses are right next to each other.

It's important to notice that the `lower()` method does **not** actually change the variable; instead it returns a **copy** of the variable in lower case. We can prove that it works this way by printing the variable before and after running `lower()`. Here's the code to do so:

```
my_name = "MARTIN"
# print the variable
print("before: " + my_name)

# run the lower method and store the result
lowercase_name = my_name.lower()

# print the variable again
print("after: " + my_name)
```

print_before_and_after.py

and here's the output we get:

```
before: MARTIN
after: MARTIN
```

Just like the `len()` function, in order to actually do anything useful with the `lower()` method, we need to store the result (or print it right away).

Because the `lower()` method belongs to the string type, we can only use it on variables that are strings. If we try to use it on a number:

```
my_number = len("Martin")
# my_number is 6
print(my_number.lower())
```

we will get an error that looks like this:

```
AttributeError: 'int' object has no attribute 'lower'
```

The error message is a bit cryptic, but hopefully you can grasp the meaning: something that is a number (an `int`, or integer) does not have a `lower()` method. This is a good example of the importance of types in Python code: **we can only use methods on the type that they belong to.**

Before we move on, let's just mention that there is another method that belongs to the string type called `upper()` – you can probably guess what it does!

Replacement

Here's another example of a useful method that belongs to the string type: `replace()`. `replace()` is slightly different from anything we've seen before – it takes two arguments (both strings) and returns a copy of the variable where all occurrences of the first string are replaced by the second string. That's quite a long winded description, so here are a few examples to make things clearer:

```
greeting = "Hi, friend!"

# replace i with o
print(greeting.replace("i", "o"))

# we can replace more than one character
print(greeting.replace("Hi", "Howdy"))

# the original variable is not affected
print(greeting)
```

replace.py

And this is the output we get:

```
Ho, froend!
Howdy, friend!
Hi, friend!
```

Notice how the first replacement has replaced both of the **i** characters – the one that's part of the word "Hi" and also the one that's part of the word "friend".

Extracting part of a string

What do we do if we have a long string, but we only want a short portion of it? This is known as taking a *substring*, and it has its own notation in Python. To get a substring, we follow the variable name with a pair of square brackets which enclose a start and stop position, separated by a colon. Again, this is probably easier to visualize with a couple of examples – let's start with a long word and extract some bits of it:

```
word = "spontaneous"

# print letters three to five
print(word[3:5])

# positions start at zero, not one
print(word[0:4])

# if we leave out the stop position it goes to the end of the string
print(word[5:])
```

print_substrings.py

and here's the output:

```
nt
spon
aneous
```

There are two important things to notice here. Firstly, we actually start counting from position zero, rather than one – in other words, position 3 is actually the fourth character[1]. This explains why the first character of the first line of output is **n** and not **o** as you might think. Secondly, the positions are **inclusive** at the start, but **exclusive** at the stop. In other words, the expression `word[3:5]` gives us everything starting at the

1 This seems very annoying when you first encounter it, but we'll see later why it's necessary.

fourth character, and stopping just before the sixth character (i.e. characters four and five).

If we just give a single number in the square brackets, we'll just get a single character:

```
word = "spontaneous"
first_letter = word[0]
```

We'll learn a lot more about this type of notation, and what we can do with it, in chapter 4.

Counting and finding substrings

A common job in programming is to count the number of times some pattern occurs in a bit of text. In computer programming terms, what that translates to is counting the number of times a *substring* occurs in a *string*. The method that does the job is called `count()`. It takes a single argument whose type is string, and returns the number of times that the argument is found in the variable. The return type is a number, so we must be careful about how we use it!

Let's use our favourite word "spontaneous" again as an example. Remember that we have to use our old friend `str()` to turn the counts into strings so that we can print them.

```
word = "spontaneous"

# count some different letters
n_count = word.count('n')
ne_count = word.count('ne')
x_count = word.count('x')

# now print the counts
print("number of n's: " + str(n_count))
print("number of ne's: " + str(ne_count))
print("number of x's: " + str(x_count))
```

count_letters.py

The output shows how the `count()` method behaves:

```
number of n's: 2
number of ne's: 1
number of x's: 0
```

Counting the number of **n**'s gives us an answer of two. Counting the number of **ne**'s gives us one, because only one of the **n**'s is followed by a letter **e**. And counting the number of **x**'s gives us the answer zero, because there's no letter **x** in the word.

A closely related problem to counting substrings is finding their location. What if instead of counting the number of **n** letters in our word we want to know where they are? The `find()` method will give us the answer, at least for simple cases. `find()` takes a single string argument, just like `count()`, and returns a number which is the position at which that substring first appears in the string (in computing, we call that the *index* of the substring).

Remember that in Python we start counting from zero rather than one, so position 0 is the first character, position 4 is the fifth character, etc. A couple of examples:

```
word = "spontaneous"
print(str(word.find('n')))
print(str(word.find('ne')))
print(str(word.find('x')))
```

find_letters.py

And the output:

```
3
6
-1
```

A couple of interesting things to notice here: when we ask for the location of the letter **n**, we get back the first position where it occurs, and when we ask for the location of a substring that doesn't exist, we get back the answer `-1`.

Of the tools we've discussed in this section, three – `replace()`, `count()` and `find()` – require at least two strings to work, so be careful that you don't get confused about the order. Remember that:

```
word.count(letter)
```

is **not** the same as:

```
letter.count(word)
```

Recap

We started this chapter talking about strings and how to work with them, but along the way we had to take a lot of diversions, all of which were necessary to understand how the different string tools work. Thankfully, that means that we've covered most of the nuts and bolts of the Python language, which will make future chapters go much more smoothly.

We've learned about some general features of the Python programming language like

- the difference between *functions*, *statements* and *arguments*
- the importance of *comments* and how to use them
- how to use Python's error messages to fix bugs in our programs
- how to store *values* in *variables*
- the way that *types* work, and the importance of understanding them
- the difference between *functions* and *methods*, and how to use them both

And we've encountered some tools that are specifically for working with strings:

- concatenation
- different types of quotes
- special characters
- changing the case of a string
- finding and counting substrings
- replacing bits of a string with something new

- extracting bits of a string to make a new string

Many of the general features will crop up again in future chapters, and will be discussed in more detail, but you can always return to this chapter if you want to brush up on the basics. The exercises for this chapter will allow you to practice using the string manipulation tools and to become familiar with them. They'll also give you the chance to practice builder bigger programs by using the individual tools as building blocks.

Exercises

Reminder: the descriptions of the exercises are brief and may be kind of ambiguous – just like requirements for programs you will write in real life! Try the exercises yourself before you look at the solutions, but make sure to read the solutions even if you find the exercises easy, as they contain extra details that may be useful.

Counting vowels

Here's the first sentence of *A Tale of Two Cities*, by Charles Dickens (who was fond of long sentences!):

```
It was the best of times, it was the worst of times, it was the
age of wisdom, it was the age of foolishness, it was the epoch
of belief, it was the epoch of incredulity, it was the season of
Light, it was the season of Darkness, it was the spring of hope,
it was the winter of despair, we had everything before us, we
had nothing before us, we were all going direct to Heaven, we
were all going direct the other way - in short, the period was
so far like the present period, that some of its noisiest
authorities insisted on its being received, for good or for
evil, in the superlative degree of comparison only.
```

What proportion of this sentence is made up of the vowels (a,e,i,o and u)? Write a program that will calculate the answer and print it out.

Hint: you can use normal mathematical symbols like add (+), subtract (-), multiply (*), divide (/) and parentheses to carry out calculations on numbers in Python.

Reminder: if you're using Python 2 rather than Python 3, include this line at the top of your program:

```
from __future__ import division
```

to make sure that division will work correctly.

Swapping letters

Here's another first sentence, this time from *Oliver Twist*.

```
emong othar public buildings in e cartein town, which for meny
raesons it will ba prudant to rafrein from mantioning, end to
which i will essign no fictitious nema, thara is ona enciantly
common to most towns, graet or smell: to wit, e workhousa; end
in this workhousa wes born; on e dey end deta which i naad not
troubla mysalf to rapaet, inesmuch es it cen ba of no possibla
consaquanca to tha raedar, in this stega of tha businass et ell
avants; tha itam of mortelity whosa nema is prafixad to tha haed
of this cheptar.
```

Unfortunately, some of the letters have got mixed up; the **a**'s and the **e**'s have been swapped[1]. Write a program that will fix this and print out the correct text (it should start: "among other public buildings in a certain town...")

Sentence lengths

Here is the opening paragraph from *The Old Curiosity Shop*:

```
Night is generally my time for walking. In the summer I often
leave home early in the morning, and roam about fields and lanes
all day, or even escape for days or weeks together; but, saving
in the country, I seldom go out until after dark, though, Heaven
be thanked, I love its light and feel the cheerfulness it sheds
upon the earth, as much as any creature living.
```

1 For this exercise I've put the entire text in lower case, just to prevent things from becoming too complicated.

The paragraph is made up of two sentences. We can see that the first sentence is short and the second one is long, but what are their actual lengths? Write a program that will calculate the lengths and print them out.

Place name, part one

Here's the first line of *Little Dorrit*:

```
Thirty years ago, Marseilles lay burning in the sun, one day.
```

It mentions the name of a city. The name of the city starts at the 19th character of the sentence and ends at the 28th character. Write a program that uses these numbers to extract just the name of the city from the sentence and print it.

Place name, part two

Using the information from part one, write a program that will calculate what percentage of the sentence is made up of the name of the city.

Reminder: if you're using Python 2 rather than Python 3, include this line at the top of your program:

```
from __future__ import division
```

Place name, part three

Using the information from part one, write a program that will print out the original sentence but with the name of the city in upper case.